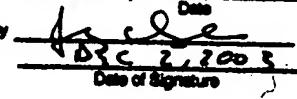


1 I hereby certify that this correspondence is being deposited as  
2 express mail # EL605496720US  
3 with the United States Postal Service in an envelope addressed  
4 to: Commissioner for Patents, P. O. Box 1450, Alexandria, VA  
5 22313-1450 on DEC 2, 2003

6   
7 Date  
8 DEC 2, 2003  
9 Date of Signature

10 Multiple Thread In-Order Issue In-Order Completion DSP and  
11 Micro-controller

12 Field of the Invention

13 The present invention is related to signal processors  
14 for use in wireless networks carrying a plurality of  
15 wireless sessions. A Digital Signal Processor (DSP) is a  
16 type of processor which typically implements an instruction  
17 set optimized for mathematical computations, but has limited  
18 input-output (IO) capability. A Micro-controller typically  
19 has extensive input-output (IO) capability, but limited  
20 computational ability. The present invention is an  
21 architecture and process for combining these two  
22 architectures in a manner best suited for wireless signal  
23 processing which typically requires both DSP and micro-  
24 controller functionality while handling multiple  
25 communications connections.

1    Background of the Invention

2           Prior art wireless communication systems are defined in  
3    IEEE protocols 802.11 and its various derivatives 802.11a,  
4    802.11b, and 802.11m. In a typical wireless communications  
5    system, an RF signal is heterodyned to an intermediate  
6    frequency and signal processing occurs to generate a stream  
7    of data forming a frame, and a device which performs this  
8    processing is known as the physical layer device (PHY) in  
9    the OSI layer definitions. The PHY acts as an interface  
10   between the RF signal and the stream of unframed data moving  
11   to the media access controller (MAC). The media access  
12   controller (MAC) layer receives unframed data and separates  
13   header information and CRC information to perform data  
14   integrity checking, producing a data stream to a host  
15   interface, where such data may be moved via a FIFO  
16   interface, or into a packet buffer whereby data is held in  
17   structures which contain pointers to the next data  
18   structure, as is typical for PCI host adapters. In a prior  
19   art system, the signal processing from an antenna to the  
20   packet memory may be called a wireless host adapter, and  
21   each processing stage of the host adapter requires  
22   specialized circuitry for the performance of each specific  
23   function. If it is desired to allow multiple simultaneous  
24   wireless sessions, which requires the user have more than  
25   one wireless host adapter, then each host adapter contains

1 its own circuitry, which performs the required PHY and MAC  
2 functions independently from any other host adapter. Each  
3 host adapter carries one wireless session, and consumes a  
4 particular amount of space and power, and each additional  
5 host adapter linearly increases the requirement for space  
6 and power. Additionally, there are several different  
7 protocols for wireless LANs, and other protocols are under  
8 development. Presently, each protocol may require its own  
9 host adapter which operates for that particular protocol  
10 only.

11 In a wireless communications system, there are often  
12 two types of processors used: a micro-controller for  
13 handling data movement to and from a host adapter memory,  
14 and a DSP to handle signal processing calculations done on  
15 incoming signals. Compared to prior art uses of micro-  
16 controllers and DSPs, the bandwidths involved in wireless  
17 communications are lower, however most modern micro-  
18 controllers and DSPs have a surplus of bandwidth available,  
19 which translates into higher power dissipation. The higher  
20 power dissipation and inseparability of the DSP function and  
21 IO function results in both types of processors being used  
22 in a typical systems, which also contributes to higher power  
23 dissipation and shorter battery.

24 In addition to the need for a hybrid DSP and micro-  
25 controller, there is also the need to be able to separate

1 processing of two channels into fixed-bandwidth processing  
2 threads. In the current art of multi-tasking real-time  
3 operating systems, multiple instances of a program are  
4 executed using separate storage contexts and a Real-Time  
5 Operating System (RTOS) which allocates a certain amount of  
6 time to each task. The overhead of an RTOS is fairly high,  
7 and context switching from one task to another takes  
8 hundreds to thousands of processor clock cycles. Because of  
9 the high overhead of context switching and the requirement  
10 of guaranteed processing bandwidth in a digital signal  
11 processor, real-time operating systems with task switching  
12 are not implemented in current DSP processors, since the  
13 processing needs to be done in something much closer to  
14 real-time and without one task blocking the others.

15 Currently, RTOS task switching is accomplished by buffering  
16 data after the task of interest is switched out of context,  
17 which means switching to an inactive state either in memory  
18 or some form of storage for recovery when the task is  
19 switched back in context at some time in the future. For  
20 this reason, a typical DSP is typically a single context  
21 performing computations, and a micro-controller handling IO  
22 uses an RTOS, and does task switching.

23 It is desired to enable a finer grained context  
24 switching which is optimized for the needs of a small  
25 plurality of channels of wireless communications links.

1 Each of these links requires processing tasks of performing  
2 DSP calculations on incoming data and moving data from one  
3 network layer to the next.

4 Figure 1 shows a prior art pipelined processor 10.  
5 Each stage performs an operation in a single stage clock  
6 cycle, although the clocks within a single stage may operate  
7 at higher rates than the stage clock. The stages are  
8 separated by registered boundaries shown as dashed lines,  
9 such that anything crossing a dashed line in figure 1 is fed  
10 through a clocked register such as a D flip flop on each  
11 clock cycle. As known to one skilled in the art, data is  
12 generally available from one stage to the next on each clock  
13 cycle, unless a condition known as "stall" occurs. In a  
14 stall condition, for example when accessing slow external  
15 memory 42, the entire pipeline receives a stall signal 46  
16 and remains in this state until data becomes available from  
17 external memory before resuming movement of data across  
18 stage boundaries. The interval of time spent waiting for  
19 external memory to become available is known as "pipeline  
20 stall time". When a pipeline stall condition occurs, all  
21 data processing comes to a halt until the stall condition is  
22 cleared, as indicated by the stall indicator signal 42.

23 In the prior art processor 10 of figure 1, a program  
24 counter 12 provides a memory address to a Fetch Address  
25 Stage 14, which passes along the address to a Program Memory

1 18 via an address buss 16. A data buss 20 returns the  
2 program data on the next stage clock to the Program Access  
3 Stage 22. The Decode stage 28 separates the data returned  
4 from program access 22 into opcodes and data, where the  
5 opcode comprises a specific instruction to perform a  
6 particular operation using either registers 24, immediate  
7 data associated with the opcode, or data memory 40. The  
8 Decode stage 28 may determine that a data value accompanying  
9 the opcode is to be loaded into a particular register  
10 location 24, or the contents of a particular register is to  
11 be rotated, etc. The decoded operation is passed to a first  
12 execution stage EX1, which may include some multiplier  
13 operations, and to a second execution stage EX2, which  
14 contains an arithmetic logic unit (ALU) 36 for performing  
15 arithmetic operations such as add, subtract, rotate, and  
16 other functions known to one in the art of processor design.  
17 Data memory 40 which is to be written or read is accessed by  
18 providing an address, and the returned data is recovered by  
19 memory access stage 38. Memory Access stage 38 is also  
20 responsible for reading and writing external shared memory,  
21 which is typically much slower than data memory 40 or  
22 register memory 26. The Write Back stage 44 writes data  
23 back to the register controller 26.  
24 The prior art processor of figure 1 performs many  
25 functions well. However, any stall condition which may

1 occur, for example, when data is read from external memory  
2 42, causes stoppage of the entire data path through  
3 assertion of the Stall signal 46 which indicates to all  
4 pipeline stages to stop forwarding information until the  
5 stall condition is cleared. For time-sensitive  
6 calculations, this stall condition can be catastrophic. It  
7 is desired to provide an architecture which allows more than  
8 one thread to simultaneously proceed through the core  
9 pipeline during a stall condition.

10

11

12 Objects of the Invention

13 A first object of the invention is a fine grained  
14 context switching processor.

15 A second object of the invention is a fine grained  
16 multithreaded processor which includes context storage for a  
17 Program Counter and a plurality of Registers.

18 A third object of the invention is a fine grained  
19 multithreaded processor which includes several stages of  
20 functionality separated by registered boundaries, such that  
21 alternating stages have access to Storage Registers, and the  
22 same set of alternating stages is using the same thread ID  
23 so that register operations for a single thread may be  
24 performed simultaneously.

1       A fourth object of the invention is a fine grained  
2 multithreaded processor which includes several pipeline  
3 stages, each stage having access to a thread ID.  
4       A fifth object of the invention is a fine grained  
5 multithreaded processor which includes several pipeline  
6 stages, each stage separated from the other by a registered  
7 boundary, and an n-way register controller which  
8 simultaneously writes data from multiple stages into a  
9 common set of registers for a particular thread ID.

10

11

12

13 Summary of the Invention

14       A processor includes a plurality of pipeline stages  
15 including a Fetch Address Stage 102 for the generation of  
16 program addresses, Program Address generators 106 and 108, a  
17 Program Access Stage 104 for receiving data 114 associated  
18 with the fetch address 112 instruction, a Decode Stage 120  
19 for decoding multiply instructions from the Program Access  
20 into individual operational actions, and passing non-  
21 multiply instructions to the ALU, a First Execution Stage  
22 (EX1) 126 receiving decoded multiply instructions and  
23 performing multiplier operations, and on cycles which are  
24 not performing multiplier operations, decoding non-multiply  
25 instructions for use by a Second Execution Stage (EX2) which

1 includes an Arithmetic Logical Unit (ALU) sending results to  
2 a Memory Access Stage (MA) 134. The Memory Access Stage  
3 (MA) 134 reads and writes results from Data Memory 150 or  
4 External Memory 152. Data which is intended for Storage  
5 Registers is handled by a Write Back to Register Stage 136.  
6 Register Storage A 140 and B 142 are controlled by a n-way  
7 register controller 138 which enables simultaneous write  
8 operations to a plurality of A and B Register locations 140  
9 and 142 during the same stage clock. The blocks which may  
10 access the n-way register controller at a single instant in  
11 time are the Decode Stage 120, the Second Execution Stage  
12 (EX2) 132, and the Write Back to Register Stage 136.

13

14 Brief Description of the Drawings

15 Figure 1 shows the block diagram for a prior art  
16 processor.

17 Figure 2 shows the block diagram for a multi-thread  
18 processor having two threads.

19 Figure 3 shows two threads of program instructions  
20 being executed by a single processor.

21 Figure 4 shows the instructions of the two threads.

22 Figure 5 shows the multi-stage progression of the  
23 instructions of figure 4 through the processor.

24 Figure 6 shows a thread-independent device decoder for  
25 addressing thread-independent devices.

1       Figure 7 shows a thread-dependent device decoder for  
2 addressing devices separated by thread ID.

3

4

5       Detailed Description of the Invention

6       Figure 2 shows a Multiple Thread In-Order Issue In-  
7 Order Completion DSP and Micro-controller 100. While there  
8 can be as many different threads as desired, for the purpose  
9 of illustration two threads A and B are shown. A thread ID  
10 162 is generated by a thread ID generator 146, which may  
11 also generate other control signals such as reset 164, which  
12 are shown on a shared control bus 160 which also carries  
13 other control signals as needed. In this manner, various  
14 stages of the Controller 100 act according to thread ID 162  
15 available to them. For example, during Thread A, Fetch  
16 Address block 102 receives a Program Counter A 106 during  
17 the A Thread, and receives a different Program Counter B 108  
18 during a second B thread. During an A thread cycle, the  
19 Fetch Address Stage 102 passes this A thread Program Counter  
20 address to the Program Memory 110, and the result is  
21 provided via a register (at dash line 116 boundary) to  
22 Program Access 104, which receives the data 114 associated  
23 with the Fetch Address 112. On the next B thread stage  
24 cycle, the Fetch Address Stage 102 receives a Program  
25 Counter B 108 and passes this address to the Program Memory

1 110. In this manner, one alternating set of Stages is  
2 successively processing A-B-A- threads, while the other  
3 stages intermixed between the alternating stages is  
4 processing B-A-B- threads. The stages of the controller 100  
5 are separated by inter-stage registers, shown as dotted  
6 lines 116, 118, 124, 154, 156, and 158. The stages 102,  
7 104, 120, 126, 132, 134, and 136 along with the inter-stage  
8 registers 116, 118, 124, 154, 156, and 158 form a micro-  
9 controller pipeline, where information from each stage is  
10 passed on to the next stage on each stage clock, and, except  
11 for stall conditions, each stage is performing operations on  
12 data within that stage and passing it on to the next. These  
13 stages are collectively known as the micro-controller core.  
14 Because the instructions are processed in sequence, the  
15 micro-controller 100 core is known as an in-order issue, in-  
16 order completion pipeline. This is only true within a  
17 particular thread, since the opposite thread may be in a  
18 stall condition, and the two threads execute independently  
19 of each other. The Program Access Stage 104 receives the  
20 executable code instructions 114 comprising instructions and  
21 immediate data, and passes them on to the Decode Stage 120,  
22 which decodes only multiply instructions, receiving register  
23 multiplication operands from register controller 138 on bus  
24 139, and passes non-multiply instructions to EX1 126 for  
25 decode. Non-multiply opcodes are received by Decode 120 and

1 passed to EX1 126 where the decode takes place, and if  
2 reading a register is required, this is done using register  
3 access bus 141. As a first example, a Load Immediate  
4 instruction comprises an opcode including a register  
5 destination and a data value to be loaded into this register  
6 destination. This load immediate instruction is passed by  
7 decode 120 to EX1 126, where non-multiply instructions are  
8 decoded. If the opcode were a load from register  
9 instruction, the decode would similarly be done in EX1 126,  
10 and register data would be delivered on bus 141 to EX1 stage  
11 126. The Write Back stage 136 is responsible for moving  
12 results back to registers A 140 or B 142. If the Thread ID  
13 is A, then the data is written into one of the A memory  
14 registers 140, and if the Thread ID is B, the data is  
15 written into one of the B memory registers 142. The first  
16 execution stage EX1 126 acts directly on data from the  
17 decode stage 120 for multiply instructions, and multiplexer  
18 148 advances multiply operand constants and register values  
19 from bus 139 from decode 120 to Second Execution Stage EX2  
20 132 via multiplexer 148. EX2 stage contains the ALU and  
21 performs arithmetic operations such as ADD, SUB (subtract),  
22 ROL (rotate left), ROR (rotate right), and other  
23 instructions commonly known in the art of Arithmetic Logic  
24 Units (ALUs). Multiplexer 148 receives multiply results  
25 from EX1 126 and register values delivered from EX1 126 via

1 register bus 141 accessing A register 140 or B register 142,  
2 depending on thread. For the case of a stall condition,  
3 there may be results of arithmetic operations stored in the  
4 EX2 ALU 132, which may be stored in a set of dedicated  
5 registers so that the non-stalled thread may continue  
6 passing results from the second execution stage EX2 132 to  
7 the memory access stage 134. Since there are two threads,  
8 and either thread may stall with computational results,  
9 there are two sets of result registers at the boundary of  
10 second execution stage 132 and memory access stage 134.

11 There are three types of data storage which are used by  
12 the processor 100. The first type of storage is Register  
13 Storage A 140 and B 142, which is very fast shared storage  
14 controlled by n-way register controller 138. The registers  
15 A 140 and B 142 are written or read simultaneously by a  
16 plurality of stages, including Decode Stage 120 for reading  
17 registers for multiply operations, First Execution Stage EX1  
18 126 for reading register values for non-multiply operations,  
19 and Write Back Stage 136 for writing results back to  
20 registers. It is useful for the alternate stages in the  
21 pipeline path to have access to registers 140 as shown with  
22 stages 120, 126, and 136 having access to register  
23 controller 138 such that when the Thread ID is A, most of  
24 the register accesses are A thread registers, and when the  
25 Thread ID is B, most of the register access are B thread

1 registers. An exception to this would be thread A having a  
2 multiply instruction in Decode 120 reading an A register 140  
3 over bus 139 while a load from register instruction on the B  
4 thread was in EX1 stage reading a B register 142 over bus  
5 141. However, most of the time, the register reads and  
6 writes tend to be on the same thread. The n-way register  
7 controller 138 allows A and B thread operations to occur  
8 simultaneously. This is important because the n-way  
9 register controller may receive simultaneously a request to  
10 write register 0 from decode Stage 120, and to read register  
11 0 from Second Stage EX2 132, or there may be a request to  
12 write back a value in Write Back 136 while there is a  
13 request to read a value in EX2 132, and data coherency  
14 requires that all of these reads and writes be handled  
15 concurrently, which requires they all be on the same thread.  
16 The n-way register controller 138 in this case furnishes the  
17 data value directly to the reading stage from the writing  
18 stage, simultaneously writing the new value into the  
19 requested register. The second type of storage memory is  
20 Data Memory 150, for which an address is provided by the ALU  
21 Second Execution unit EX2 134, and is available to the  
22 Memory Access Stage 134. The Register Storage 140 and 142  
23 has the highest memory bandwidth, since it must be able to  
24 write and read multiple registers during a single stage  
25 cycle, while the Data Memory 150 is only able to read or

1 write a single address during a single stage cycle. The  
2 External Memory 152 is potentially the slowest memory, since  
3 the processor 100 may be competing with other devices for  
4 this bandwidth over a shared bus. When slow external memory  
5 152 is read, and the read lasts more than a single stage  
6 clock cycle, a "pipeline stall" condition occurs, and no  
7 data is moved through any of the stages described until the  
8 stall condition is removed. Often, in shared resource  
9 systems, a bus controller 164 controls stall signal 166  
10 where the bus controller receives a "/REQUEST" signal  
11 indicating a request to start a transaction, and the bus  
12 controller replies with an "/ACKNOWLEDGE" signal indicating  
13 availability of the device to accept the data. For write  
14 cycles, it is possible for the controller to simply store  
15 successive data cycles in a fifo for dispatch over time when  
16 the memory device becomes available, as is known in the art  
17 as "write caching", which may prevent pipeline stall  
18 conditions. However, data read conditions often cause a  
19 stall during the interval between the /REQUEST of the bus  
20 controller 164 to read remote data and /ACKNOWLEDGE  
21 associated with availability of data by the remote device.

22 The Thread ID 162 indicates whether an A or B thread  
23 cycle is being executed, and it is also readable by the  
24 program so that a particular thread is able to determine  
25 whether it is running on the A or the B thread. A single

1 instance of the program to execute may then be stored in  
2 Program Memory 110, and each separate instance of the  
3 program may read the thread ID to determine whether it is an  
4 A or a B thread. This is particularly useful in a wireless  
5 system with two communications channels by having each  
6 thread separately handle each wireless channel. There are  
7 several advantages of this approach. From the earlier  
8 description, it can be seen that a stall condition such as  
9 reading from external memory causes the entire processing  
10 sequence to halt. In a typical multi-threaded RTOS, the  
11 overhead of swapping context memory means that hundreds or  
12 thousands of instructions are executed before the overhead  
13 intensive context switch occurs. This is done infrequently  
14 to avoid "thrashing", where much of the processor time is  
15 spent changing contexts, and very little is spent handling  
16 actual processes. By switching threads on alternating  
17 cycles, two clear advantages are accrued. The first  
18 advantage is that by interleaving stages requiring register  
19 access through controller 138, the stages are able to  
20 instantaneously access register values and achieve data  
21 coherency and data consistency in these accesses. The  
22 second advantage is that the interleaving of cycles allows  
23 the computational results of a stage handling any given  
24 thread (A or B) to be simply passed on to the following  
25 stage on the following stage cycle without the requirement

1 for each stage to keep thread context storage for each  
2 thread, thus reducing inter-stage storage requirements. By  
3 contrast, if all of the stages were simultaneously given the  
4 same context value (all stages 102, 104, 120, 126, 132, 134,  
5 136 simultaneously processed thread A, followed by these  
6 same stages simultaneously processing thread B), the  
7 controller of figure 2 would also work satisfactorily and  
8 handle two threads independently, however the intermediate  
9 results for each stage would have to be placed in temporary  
10 thread context storage, and then retrieved for the following  
11 thread cycle. While the processing bandwidth would be the  
12 same as shown in figure 2, the inter-stage memory  
13 requirements would be significantly higher. It is therefore  
14 believed that alternating thread IDs across successive  
15 stages such that stages FA 102, Decode 120, EX2 132 and WB  
16 136 are handling one thread ID while stages PA 104, EX1 126,  
17 and MA 134 are simultaneously handling the other thread ID  
18 is best mode as shown in figure 2.

19 The granularity of information moved from stage to  
20 stage is established by a stage clock (not shown) which  
21 controls the simultaneous transfer of information from stage  
22 to stage across inter-stage boundaries using registers 116,  
23 118, 124, 154, 156, and 158 shown as dashed lines. These  
24 registers are synchronously clocked registers as known to  
25 one skilled in the art. The thread ID alternates values

1 between A and B on successive stage clock cycles. When a  
2 stall condition occurs, the signals Stall A and Stall B 160  
3 are distributed to each stage so they may suspend further  
4 processing of that particular thread until the stall  
5 condition is removed. The non-stalled thread continues to  
6 execute without interruption or reduction in performance,  
7 which is one of the principle advantages of the multi-  
8 threaded micro-controller. The requirement for the non-  
9 stalled thread to continue operation while the stalled  
10 thread waits for external memory 152 availability results in  
11 thread-specific storage at each stage boundary 116, 118,  
12 124, 154, 156, 158, however the amount of thread information  
13 stored in stage registers is much smaller than the entire  
14 thread context as would be stored in the prior art of figure  
15 1 with a multi-tasking operating system.

16 Figure 3 shows an elapsed time view of the two threads,  
17 where a single instruction from each thread is shown passing  
18 through the various stages of the microcontroller. Thread A  
19 200 represents the movement of a MUL (multiply) instruction  
20 204 followed by an ADD instruction 208 passing through the  
21 micro-controller. In this view, the instructions are  
22 separated by thread as thread A 200 and thread B 202 for  
23 clarity, although it is understood that while the  
24 instructions are shown starting at the same time, the  
25 instructions shown in thread A and thread B are completely

1 unrelated to each other in timing, and may be displaced many  
2 cycles from each other. The MUL instruction 204 passes  
3 through the FA, PA, DEC, EX1, EX2, MA, and WB stages,  
4 followed by the ADD instruction 208, which also passes  
5 through these same stages in sequence, as shown on Thread A  
6 200. Thread B 202 shows a SUB (subtract) instruction 206  
7 followed by a ROL (rotate left) instruction 210 passing  
8 through the same stages FA, PA, DEC, EX1, EX2, MA, WB in  
9 sequence. Each stage FA, PA, DEC, EX1, EX2, MA, WB is  
10 operating in succession on thread A and then thread B, and  
11 the two threads are fully asynchronous. As discussed  
12 earlier, a stall condition on thread A has no effect on the  
13 processing of instructions in thread B.

14 Figure 4 shows a sequential set of instructions for  
15 thread A 216 and a set of sequential instructions for thread  
16 B 218, the first two of which were discussed in figure 3.  
17 In practice, the program memory 110 of figure 2 holds these  
18 instructions, which begin to execute on both threads A and B  
19 upon release of a reset signal, and the two threads may be  
20 executing the same program. At some point, each instance of  
21 the program may contain instructions which query the thread  
22 ID 146, and branch to one task if the thread ID is A, and  
23 branch to a different task if the thread ID is B. In this  
24 manner, a single program residing in program memory 110 may  
25 be used to execute multiple threads of a single program.

1 The advantage of using multiple instances of the same  
2 program which are each examining thread ID 162 is that  
3 multiple instances of a single program require only a single  
4 stored version, reducing the size requirement of program  
5 memory 110.

6 Figure 5 shows the progression of the set of  
7 instructions shown in figure 4 through the processor 100.  
8 The MUL, ADD, MOV, ROR instructions of figure 4 thread A are  
9 shown as they progress through the FA, PA, DEC, EX1, EX2,  
10 MA, and WB stages. Any single stage such as FA executes  
11 alternating thread A and thread B instructions, and each  
12 instruction progresses to the next stage.

13 Figure 6 shows an example of decoding a thread-  
14 independent memory access using only the top 3 bits of an  
15 address. A 32 bit address A0-A31 250 such as may be  
16 generated by Memory Access stage 134 of figure 2 is  
17 presented to external memory 152 of figure 2. The top 3  
18 address bits A31-A29 are provided to the address input of a  
19 decoder 252, and for each combination of these 3 address  
20 bits, one of the device outputs 0-7 is asserted. For  
21 example, hex addresses in the range 0x00000000 to 0x1fffffff  
22 could cause a select for device 0 254 to be asserted, and  
23 addresses in the range 0x20000000 to 0x3fffffff could cause  
24 a select for device 1 256 to be asserted. Each memory  
25 mapped device controlled by this decoder such as a static

1 memory device, etc would have its address lines tied to A0-  
2 A28 for all such devices, while each device would have one  
3 enable line tied to a device decode such as Device 0 254,  
4 Device 1 256, etc. Read/write lines for the device would be  
5 driven by control lines from memory access 134 as known to  
6 one skilled in the art. The decoder of figure 6 would be  
7 used for thread-independent devices, such that the read and  
8 write activities from thread A and thread B decode to the  
9 same device. This functionality is useful where the thread  
10 is interested in checking the status of a device, or where  
11 the threads use their own thread knowledge to separate  
12 addresses. For example, the program of thread A could use a  
13 range of memory exclusive of the program of thread B.

14 Figure 7 shows the case of a device decoder for thread-  
15 specific devices, where each device only responds to a  
16 particular thread. A 32 bit address A0-A31 250 such as may  
17 be generated by Memory Access stage 134 of figure 2 is  
18 presented to external memory 152 of figure 2. The top 2  
19 address bits A31-A30 are provided to the address input of a  
20 decoder 260, and for each combination of these 2 address  
21 bits plus the thread ID bit 262, one of the device outputs  
22 0-7 is asserted. For example, for Thread ID = A, hex  
23 addresses in the range 0x00000000 to 0x3fffffff could cause  
24 a select for thread A device 0 264 to be asserted, and  
25 addresses in the same range 0x00000000 to 0x3fffffff for

1    thread ID = B could cause a select for thread B device 0 266  
2    to be asserted. Each memory mapped device controlled by  
3    this decoder such as a static memory device, etc would have  
4    its address lines tied to A0-A29 for all such devices, while  
5    each device would have one enable line tied to a device  
6    decode such as thread A Device 0 264, thread B Device 0 266,  
7    etc. Read/write lines for the device would be driven by  
8    control lines from memory access 134 as known to one skilled  
9    in the art.

10      A memory map describes the mapping of addresses to  
11    devices. Figure 6 described a way to perform thread-  
12    independent device decodes, where each device occupies a  
13    space in a memory map accessable by all threads, and figure  
14    7 describes a way to perform thread-dependant device  
15    decodes, such that the memory map is described for a  
16    particular thread.

17      The sizes of the various addresses and data paths may  
18    vary greatly depending on the particular application,  
19    however, it may appear that for wireless applications and  
20    others, an address size of 32 bits and a data size of 16  
21    bits provides good performance. While these sizes are not  
22    intended to limit the scope of the invention as set forth,  
23    an address size of 32 bits and data size of 16 bits would  
24    allow a set of 16 registers A 140 and B 142 to be used in  
25    pairs to form 32 bit address pointers for indirect

1 addressing, as known to one skilled in the art. A 32 bit  
2 address would also imply that the decode stage 120 may get  
3 an indirect relative 16 bit address offset on a single  
4 cycle, or it may wait for a full 32 bit address on two  
5 cycles. For this reason, the decode stage is shown as one  
6 or two cycles. For two cycle operations, the following  
7 cycle is not a multiply operation, so the EX1 stage may be  
8 bypassed. In this manner, the execution length of the  
9 instruction may be preserved. This may require additional  
10 storage at the decode stage such that the full 32 bit  
11 address may be passed through the pipeline on the next same-  
12 thread operation.

13 Other registers duplicated on a per-thread basis and  
14 known to those skilled in the art of processor design are  
15 not shown for clarity. It is clear that these registers  
16 would also be duplicated for each thread, and could either  
17 be part of the registers A 140 and B 142, or Program  
18 Counters A 106 or B 108, or may be present in one of the  
19 associated stages. One such other register known to those  
20 skilled in the art is a Stack Pointer which is used for  
21 returning from subroutines to re-establish register state  
22 prior to the jump to subroutine. Another such register is a  
23 status register for keeping track of the result of  
24 arithmetic and multiplicative operations, as well as  
25 interrupt status. Another set of registers may be used for

1 looping operations, and are known as a HWS register to store  
2 the program counter A 106 or B 108 during loop operations,  
3 an LA register for storing loop addresses, and LC register  
4 for keeping track of loop iterations. Each of these  
5 registers would be duplicated for A thread and B thread such  
6 that each thread has all of the resources required for a  
7 single thread, while using as much duplicated hardware as  
8 possible.